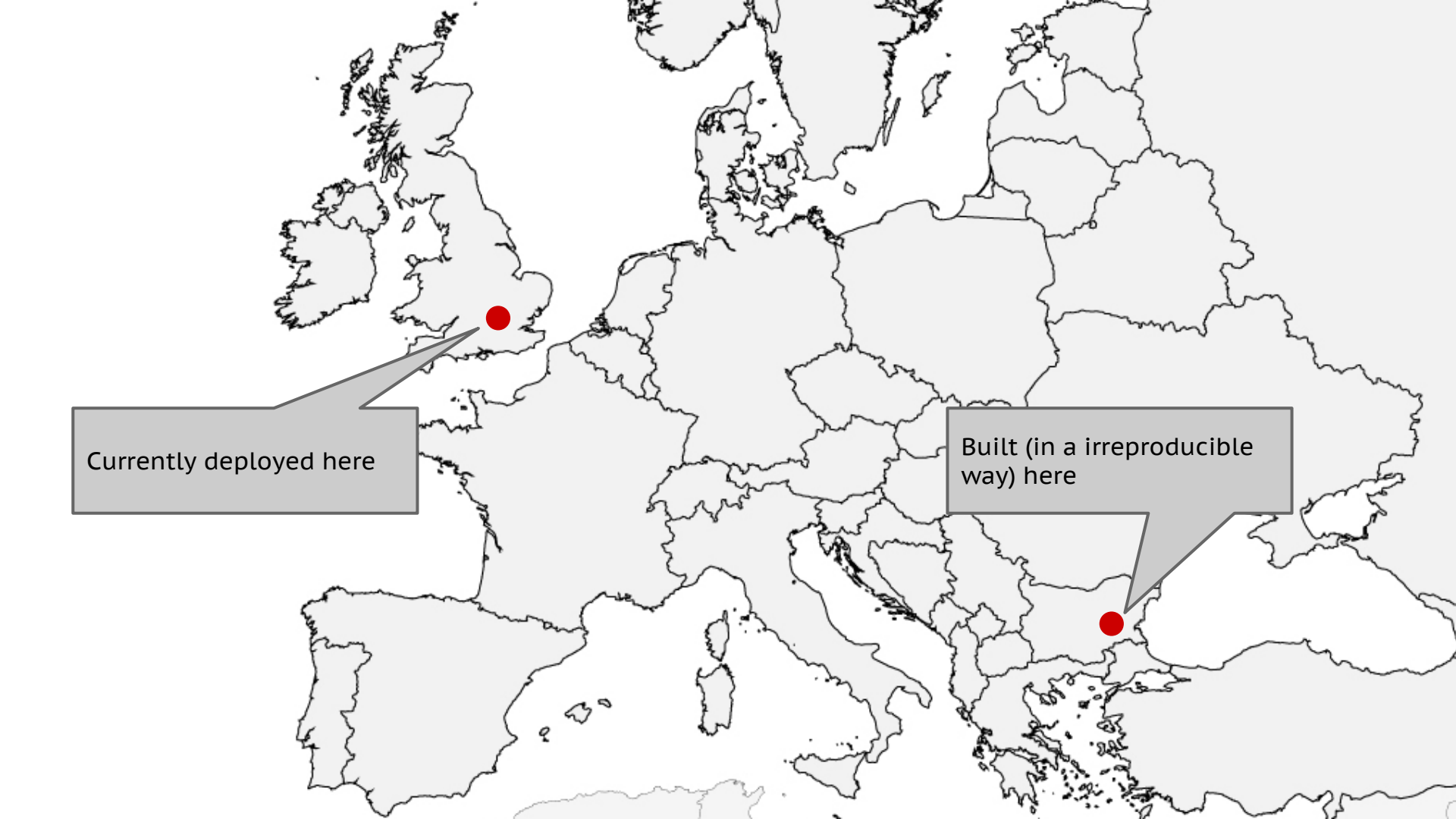
The Machine is dead, long live the Machine

Service resilience and deployment automation at the BBC

I am Yavor Atanasov or Явор Атанасов if you read Bulgarian

Today I will be speaking with you as a software engineer from BBC’s central engineering team.


Currently deployed here
Built (in a irreproducible way) here

The BBC

Depth of audience and breadth of services

- 5th largest site in UK, 55th globally
- Top 20 in News, Sport, Arts, Kids and Teens

Source: Alexa

Our team helps others deliver software fast and reliably - now and in the long term

We provide tools used for building and deploying services on AWS

What are we speaking about today?

- Where we were **before**
- Where we are **now**
- **Principles** we've followed
- **How** we followed them
- What does it all **mean**

Our previous platform

- Shared tenancy of services
- Strong **separation** of Ops and Dev
- Devs deploy to test, Ops deploy to live
- **Limited** set of technologies

What this means for people

- Long release cycle - **harder risk management and frustration**
- Separation of OPS and Dev - **friction and alienation**
- Limited technologies - **forcing people to cut an apple with a chainsaw**

What this means for our services

- **Higher risk** of breaking due to updates
- **Slower resolution** of live issues
- **Wider impact area** in case of a problematic release

The need for change

- Better service **isolation**
- Faster, more **continuous deployments**
- Easier **infrastructure** provisioning
- **No** hard **limit** on technologies
- Let **Dev** and **OPS** eat each other’s dog food

Brand new world

Continuous Delivery, Cloud, DevOps

Freedom and responsibility

- Services have dedicated infrastructure
- Teams can create infrastructure
- Teams control their deployment cycles
- Teams choose their technologies
- Teams are responsible for their service

How are we doing in numbers

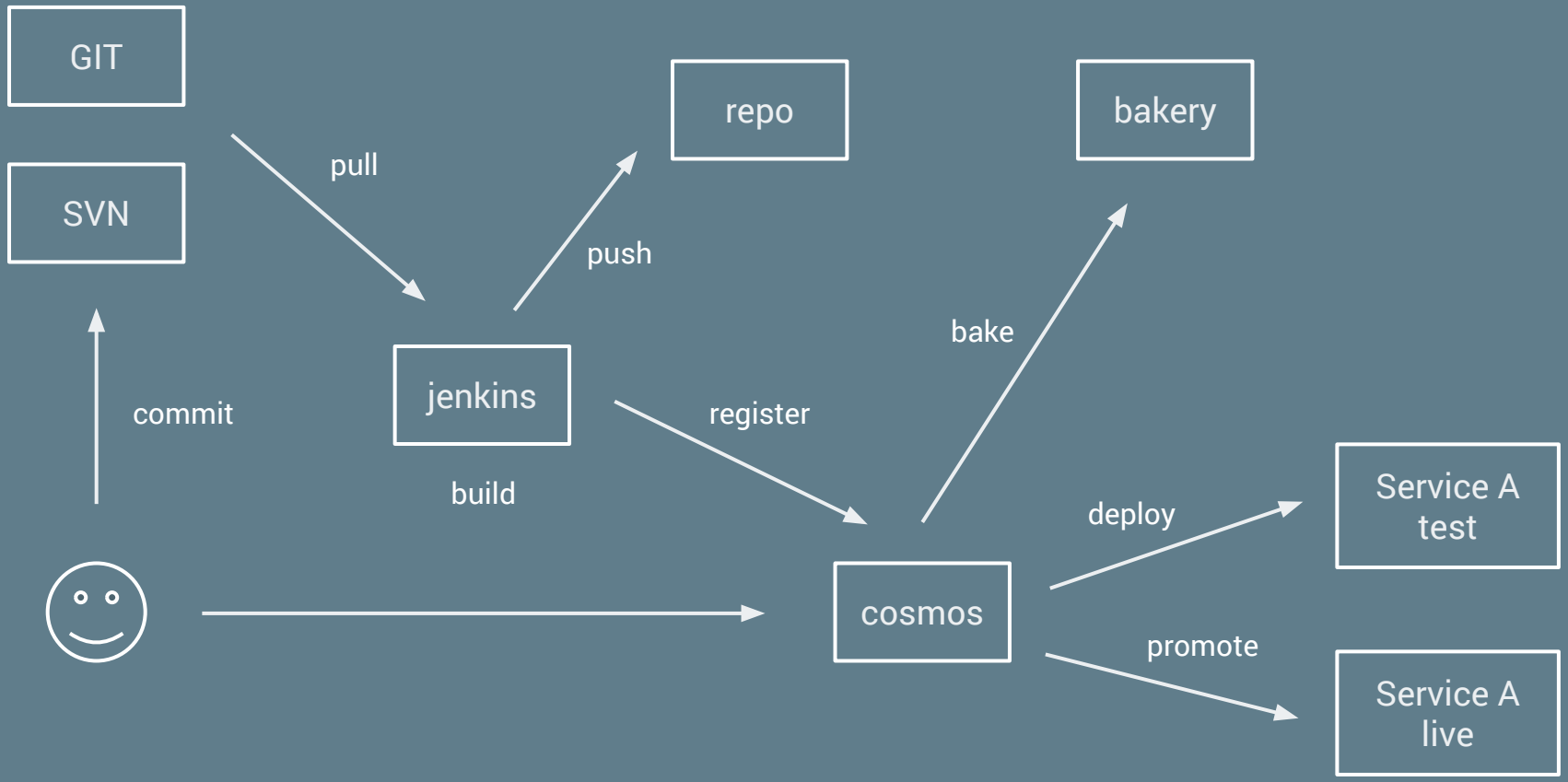
- > 300 deployments per day
- 60,000 Deployments in first 18 months
- Time to live from 2 days to 10 minutes

Who is using our tools and services

- All key video transcoding/packaging for BBC iPlayer
- Pipeline delivering election results to BBC News
- Live text for all BBC Sport events
- … a lot more

How does it look like

The path from keyboard to audiences


GIT
SVN
pull
repo
push
bakery
commit
jenkins
build
register
bake
deploy
Service A
test
cosmos
promote
Service A
live

Building your service

Building blocks

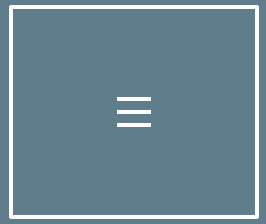
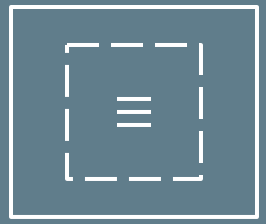
source

package

container

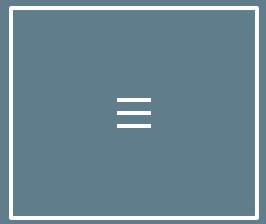
machine



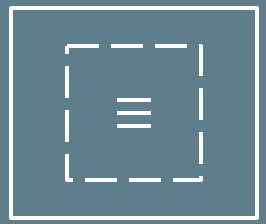



Source built into a package installed on a machine

Source built into a package installed in a container put on a machine



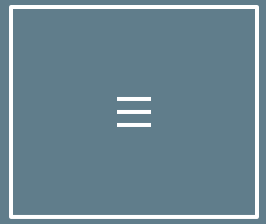
Source put on a machine directly



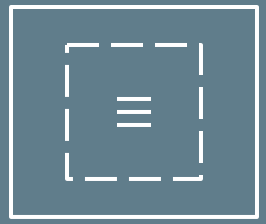
Source put in a container put on a machine

A.

B.

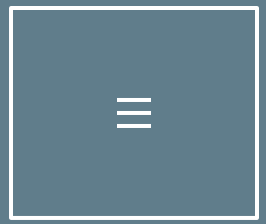


C.

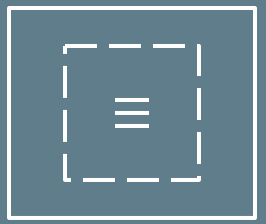


D.

What is wrong with these?



C.

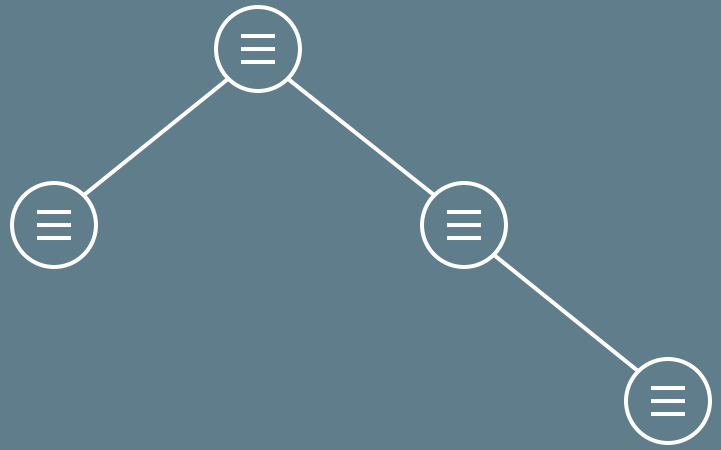


D.

How do you express dependencies of your software without packaging?

...don’t tell me bash scripts

Leave dependency management to systems designed for that!



Whether you use containers or deploy directly onto a machine - package your code

That leaves us 2 options

A.

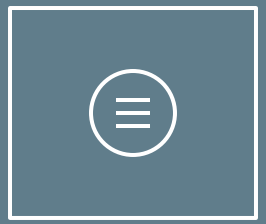
B.

Deploying (docker) containers is a dynamic topic

- Deployment and management tools are still immature
- Dependency management between things in the container and the underlying host is an open topic (fat vs thin containers)
- Cloud vendors may need to think about billing based on container usage if they want to abstract completely the underlying virtual instances
- etc...

c.

So for now our service is this ...



A.

source → package → machine

We need to build two binaries

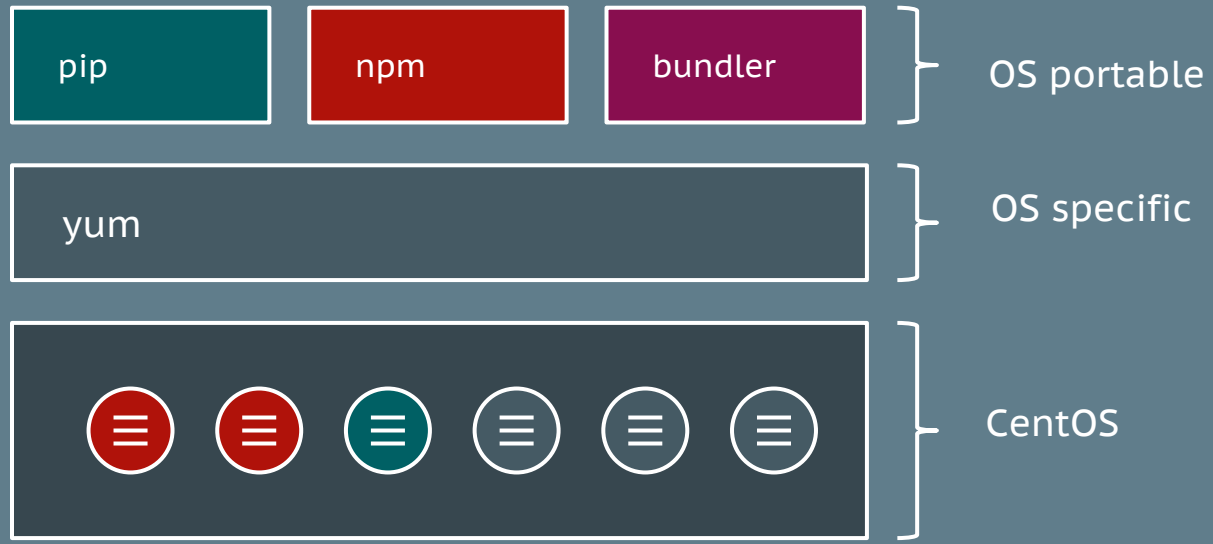
- Packages - e.g. **RPM**, **DEB**, etc.
- Machine images- e.g. **AMI**

Important principles

- Build binaries in a reproducible way
- Build them once
- Leave dependency management to tools made for it

Building a package

Different levels of package managers



Which manager to package for?

If all your dependencies come from one package manager - package for that.

e.g. I only depend on node modules, so I'll build an npm package

Dependencies across package managers?

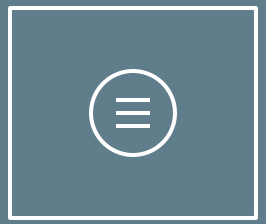
If you have dependencies stretching across different managers - package for the lower level one, resolve all higher level dependencies at build time and include them in the package.

CI is critical, choice of CI is irrelevant

Building within clean chroots using `mock` or within `docker` containers ensures all your build dependencies are specified correctly and your build is reproducible.

How we build packages

- We use Jenkins
- We build within chroots using `mock`
- Our services are built as RPM packages

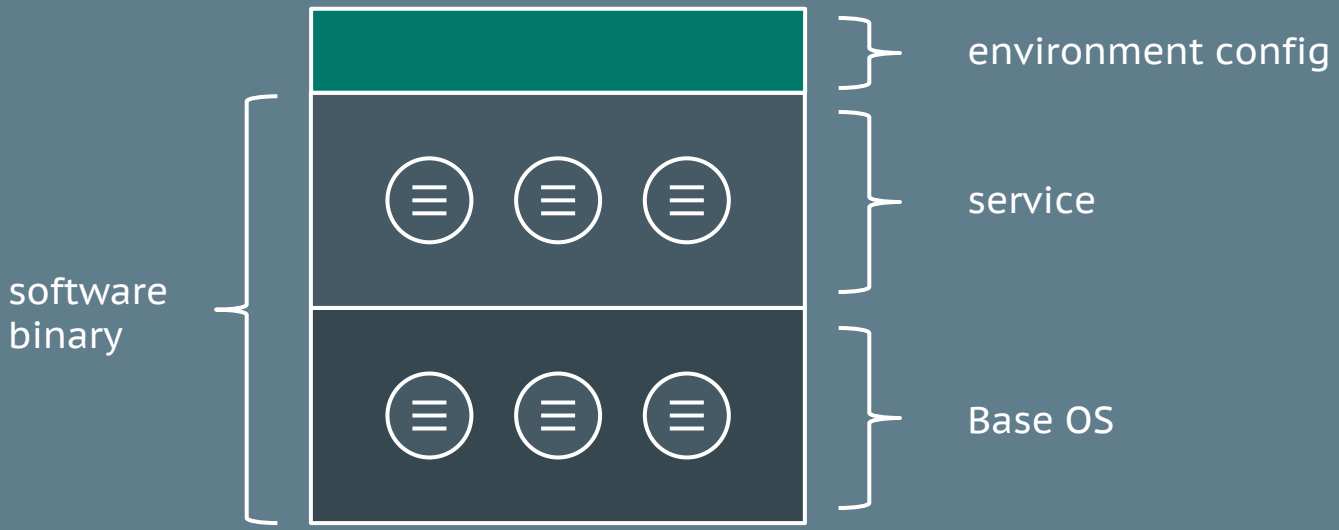


Building the machine

Important principles

- Build binaries in a reproducible way
- Build them once
- Leave dependency management to tools made for it

What's in a machine image?



Machines and environments

Keep the binaries the same, do the minimal change - configuration

Bake it all vs bootstrap

full bake

bootstrap

Image per service per environment

no startup dependencies

build binaries once

Just one base image

network dependencies on startup

building your service every time on every instance

Bake it all vs bootstrap

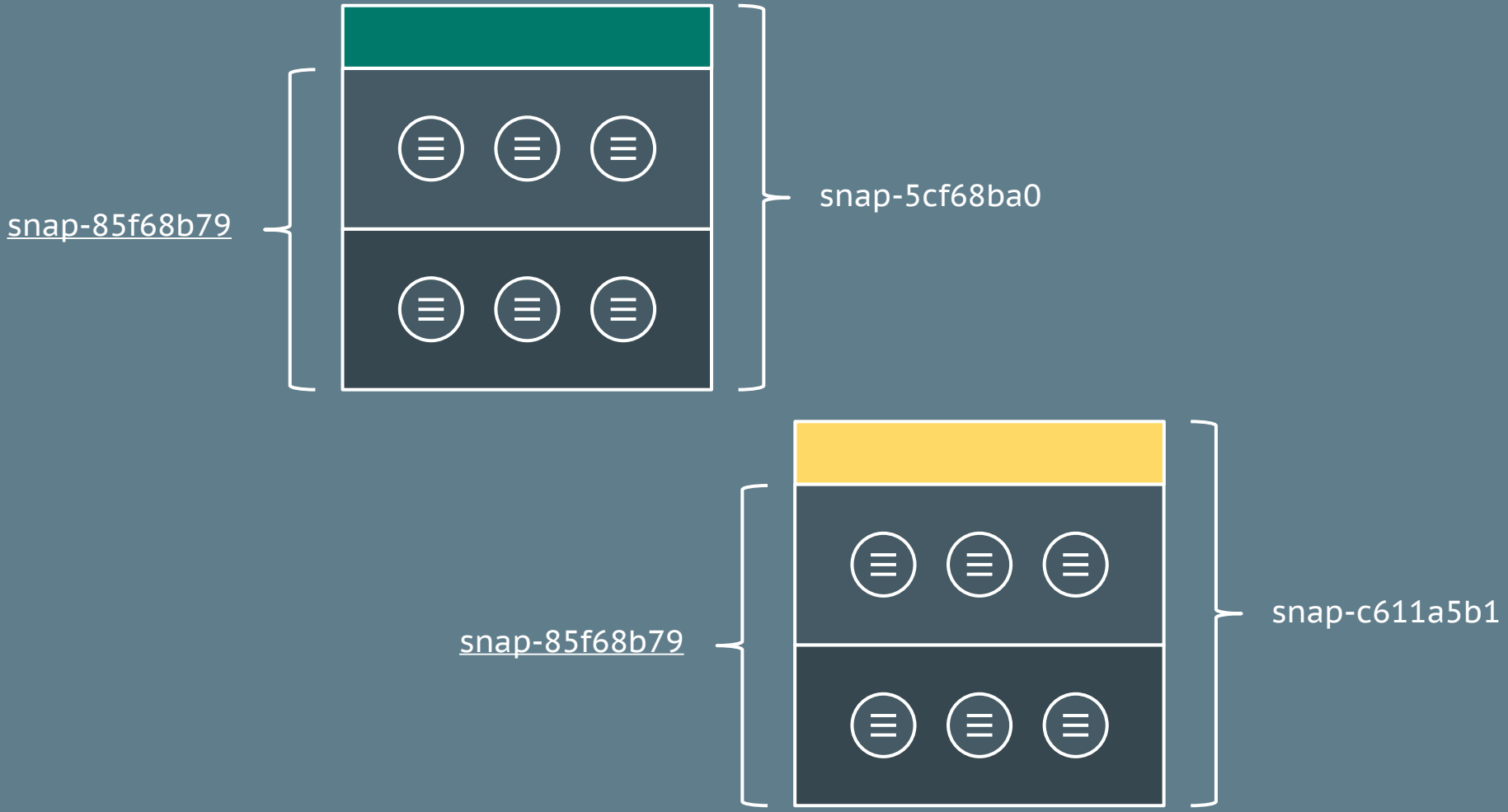
- Think about network dependencies on startup
- Think about startup time and reliability
- Think about building binaries once

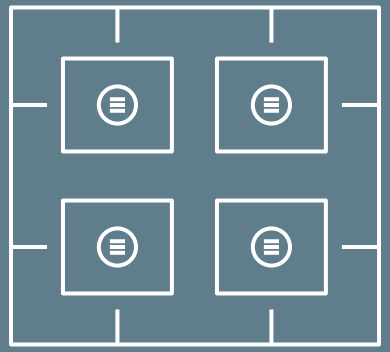
How we bake Amazon Machine Images (AMIs)

Our Bakery

- Takes repositories information, packages to install and environment specific configuration
- Bakes AMIs using a 2 step snapshot process – 1 snapshot just for the software and 1 for the software with the configuration

2 step snapshotting


snap-85f68b79
snap-5cf68ba0
snap-85f68b79
snap-c611a5b1



Provisioning cloud infrastructure

Hardware is now software, let’s treat it as such…

Important principles

- Build binaries in a reproducible way
- Build them once
- Leave dependency management to tools made for it

Infrastructure as code and AWS CloudFormation

- Manages infrastructure dependencies
- Handles underlying AWS API interactions
- Supports rollbacks
- Reproducible
- We can version infrastructure with our code

What does that mean for my service?

- I can build identical copies of my service in different environments
- I can version my infrastructure templates with my code and reproduce the full stack at any point in time

So my application is not just software, it is software and infrastructure combined

How we provision infrastructure

- We define stateless and stateful infrastructure in separate templates
- We use CloudFormation abstraction libraries to programmatically generate templates - (e.g. https://github.com/cloudtools/troposphere)

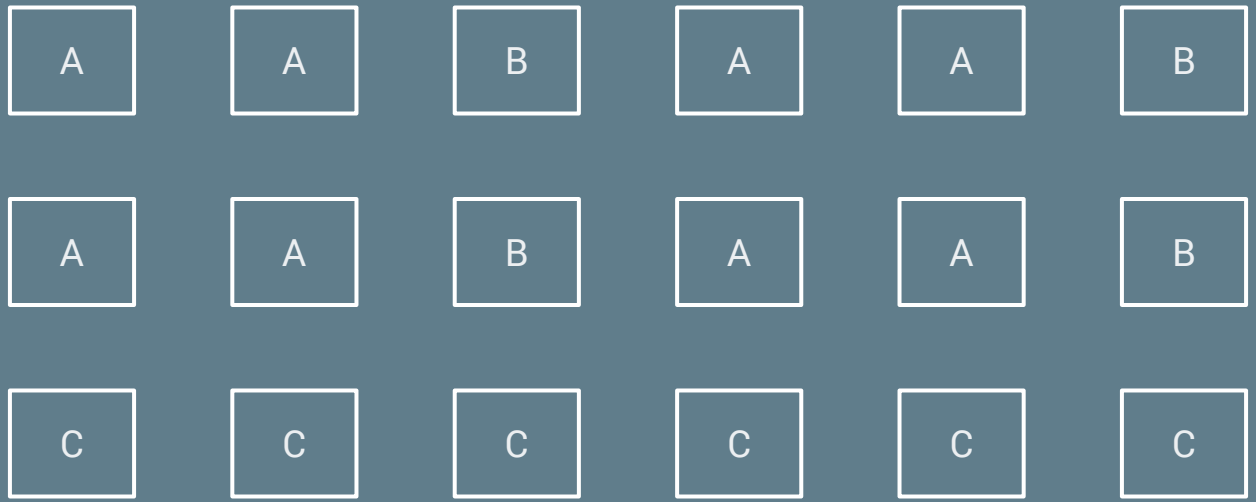
Cloud architecture

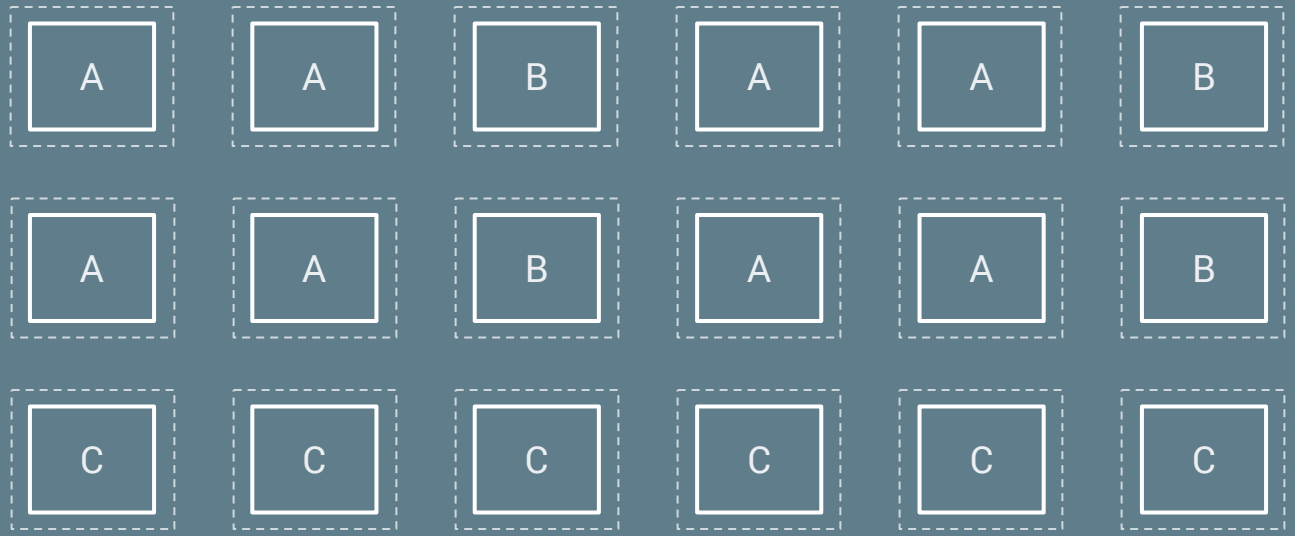
Architectural considerations

- Resilience and security through levels of isolation
- High availability through multiple zones and regions
- Scalability

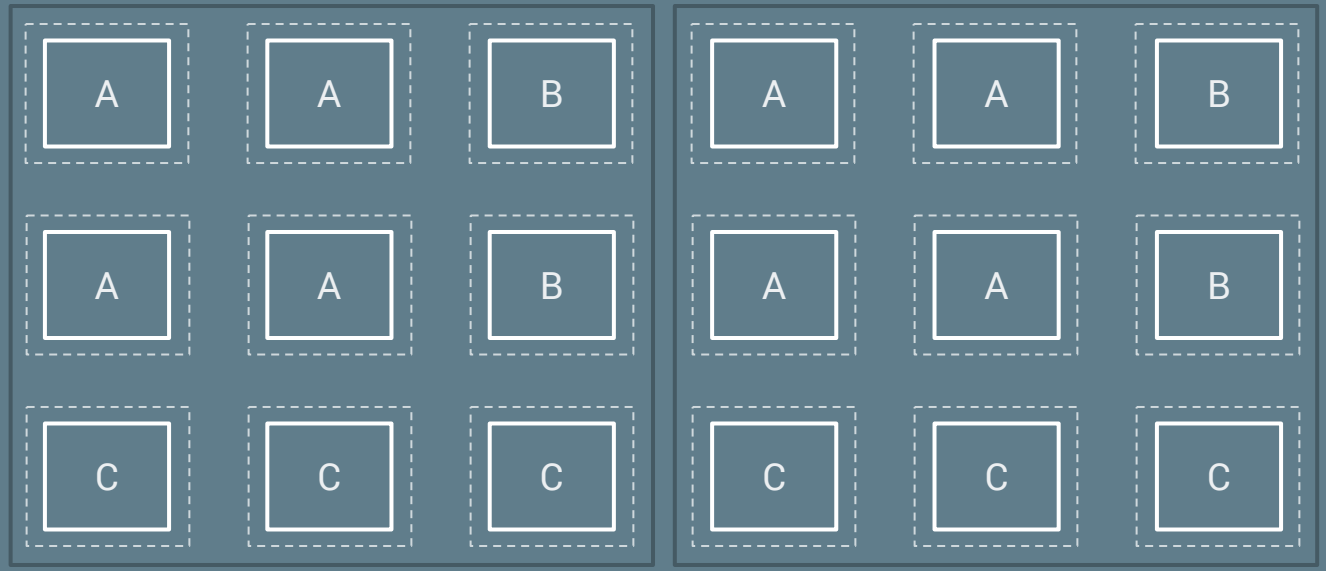
Levels of isolation

- Network and instance access - be isolated by default
- Resource isolation - find all API limits and resource limits of your cloud vendor and avoid sharing those among your critical services. The *cloud* is finite!


A A B A A B
A A B A A B
C C C C C C

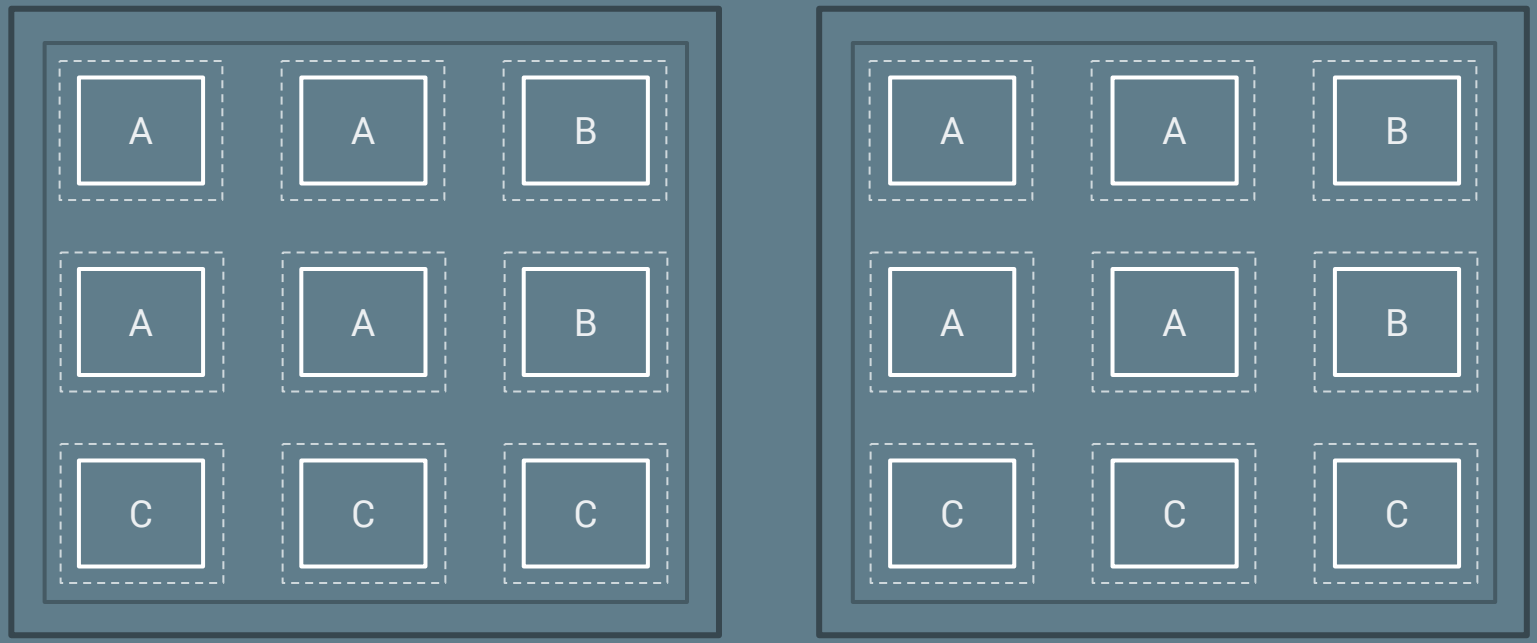


security groups to isolate instances



security groups to isolate instances

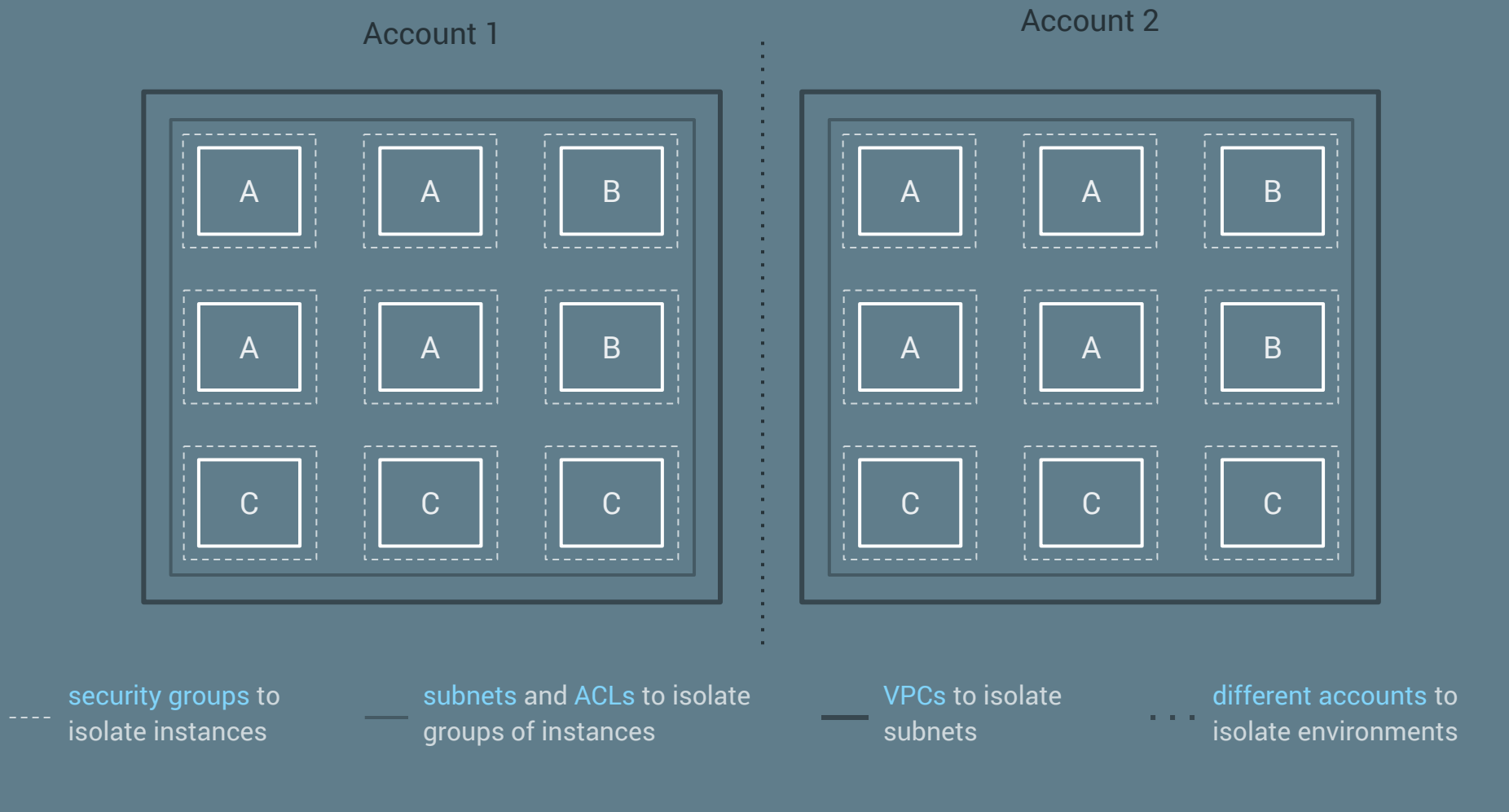
subnets and ACLs to isolate groups of instances



security groups to isolate instances

subnets and ACLs to isolate groups of instances

VPCs to isolate subnets

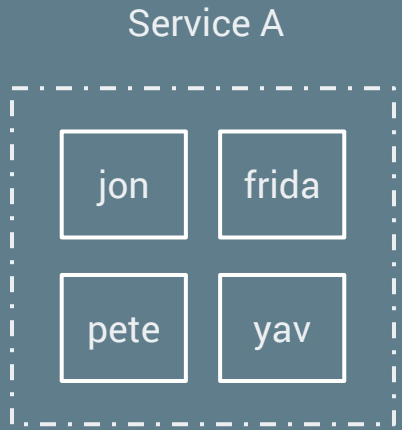

Account 1
A
A
B
A
A
B
C
C
C
Account 2
A
A
B
A
A
B
C
C
C
security groups to isolate instances
subnets and ACLs to isolate groups of instances
VPCs to isolate subnets
different accounts to isolate environments

Scalability and resilience

If you are not using AutoScaling Groups, you are doing it wrong!

- ASGs ensure your instances are always running
- ASGs handle multi AZ for you
- ASGs make you a better engineer
- ASGs can deploy services

Forget about your instances, they are just a unit of computational capability



do not give your instances names...

...think of them as threads running your service...

...that you can increase if needed

Chaos Monkey can help you kill the machine from your mind

Don’t say: *Chaos Monkey kills my instances*

Instead say: *Chaos Monkey impacts my capacity*

AutoScaling Groups help us deploy services

- Define your update policy
- Bake an AMI
- Update the ASG image id
- Watch your service rolling forward

Putting all this together

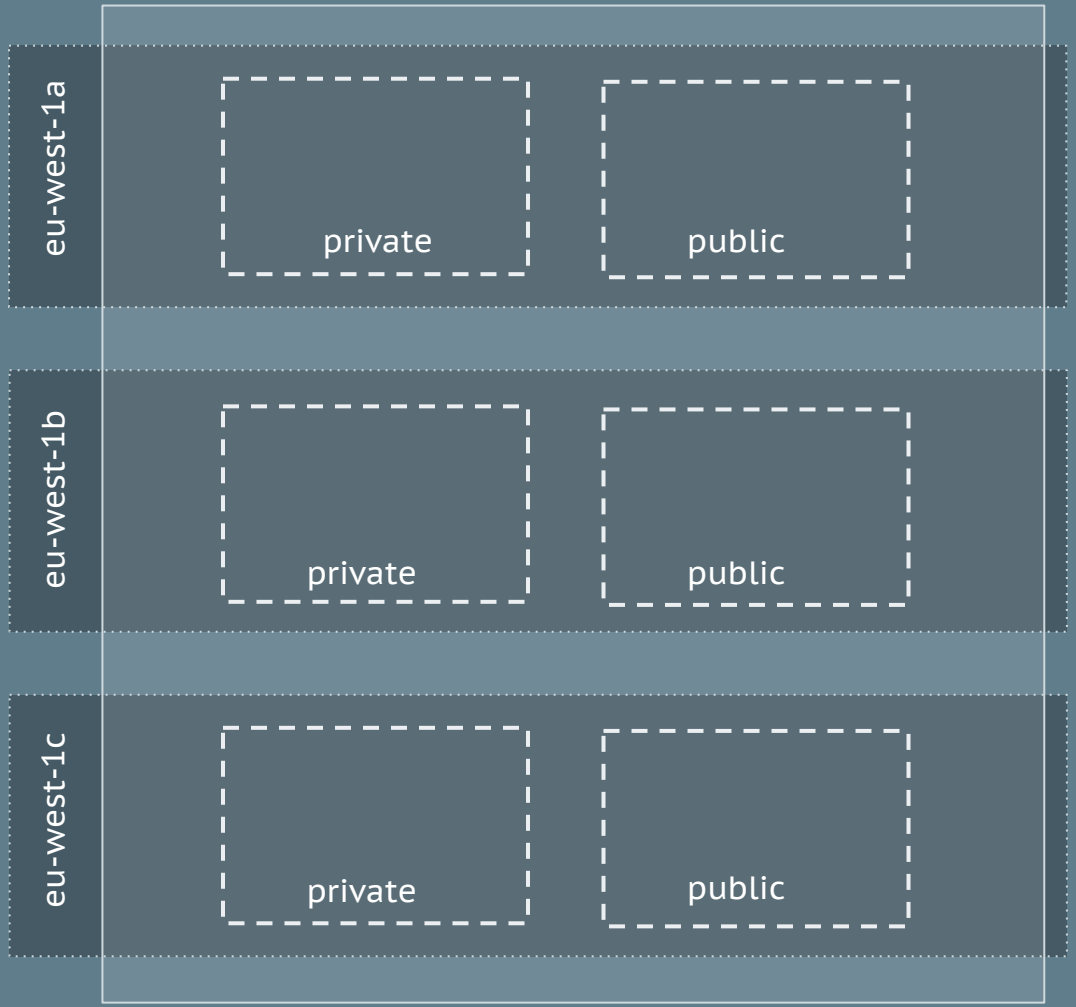
- Use VPCs, subnets and ACLs to isolate network access to your services
- Always spread your infrastructure to at least multiple AZs
- Use multiple AWS accounts for different services and/or environments
- Run your service in ASGs

High level overview of our infrastructure

Service specific infrastructure

Teams can create any infrastructure, we help with defaults

service = AutoScalingGroup
Security Groups
IAM Roles and Policies
Elastic Load Balancer
Route 53 Record
+
RDS databases
SQS Queues
S3 Buckets
etc...



Core infrastructure

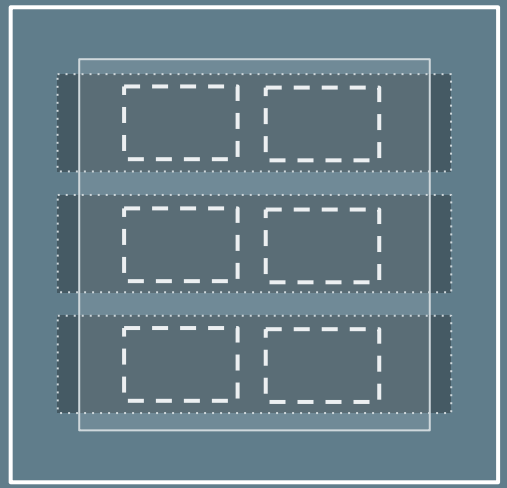
VPC with private and public subnets in the eu zones

Provides the frame upon which services' infrastructure is built

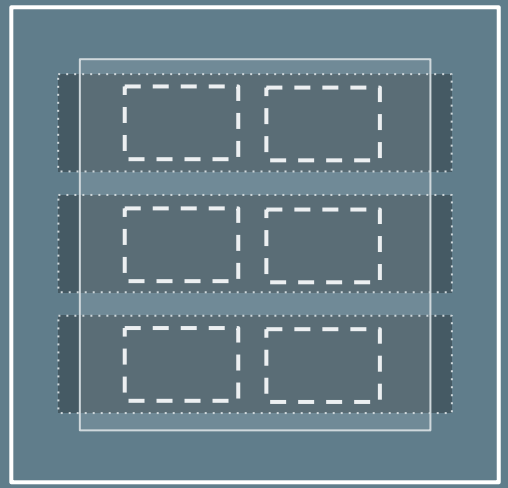
Provides security and resilience through levels of isolation

We use many AWS accounts

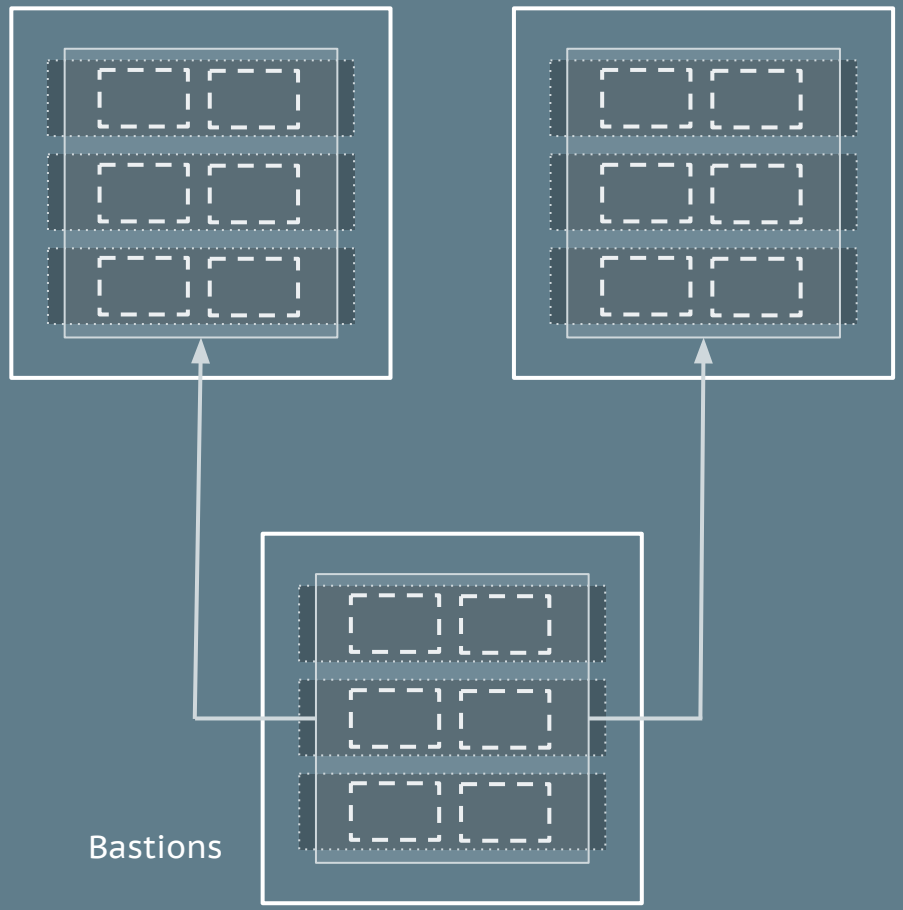
Account X



Account Y



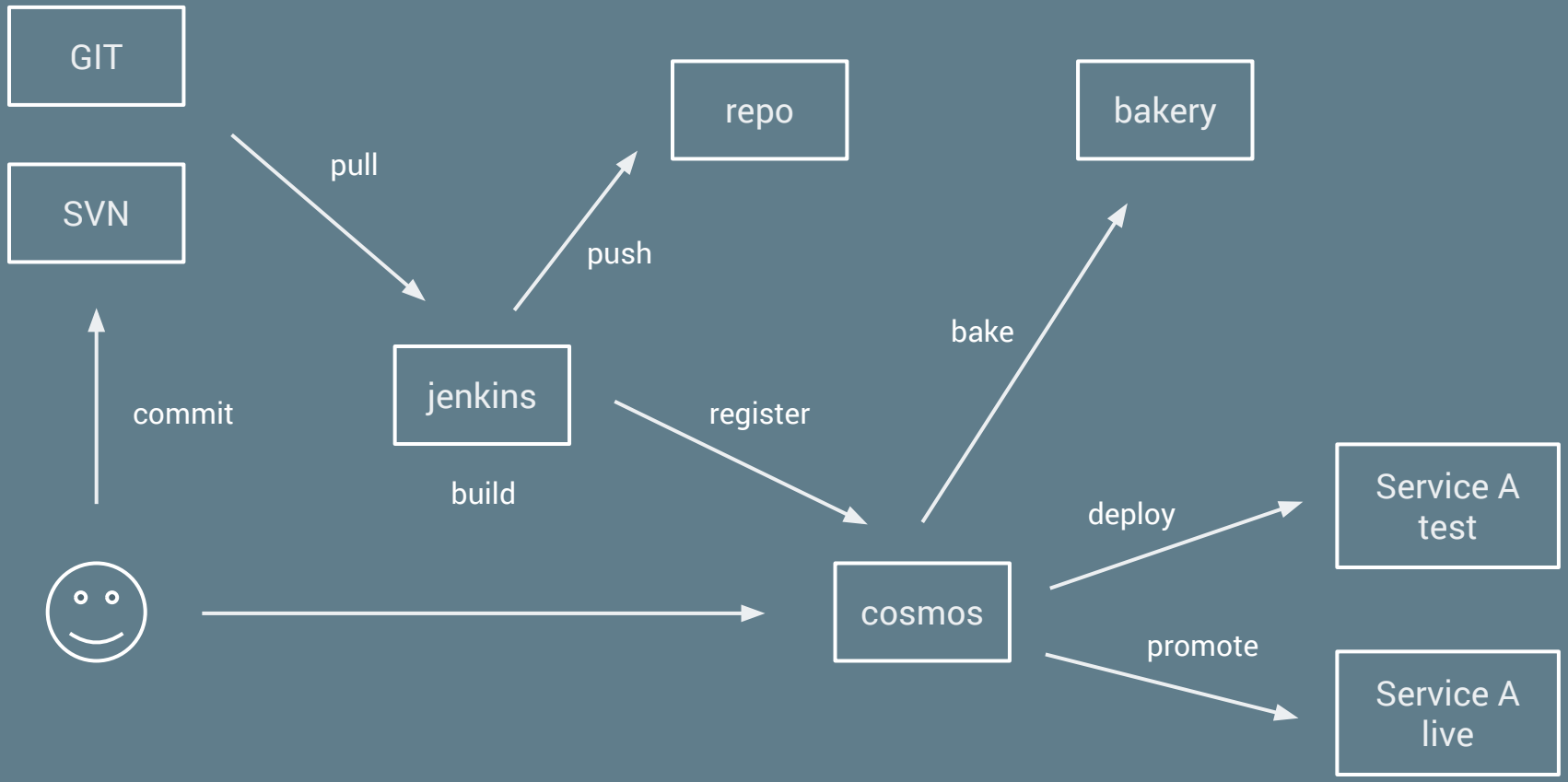
...



VPC peering and SSH access

We peer all our VPCs with the VPC of our central bastion service in order to provide SSH access to developers to their services

Let’s look at our deployment pipeline again


GIT
SVN
pull
repo
push
bakery
commit
jenkins
build
register
bake
deploy
Service A
test
cosmos
promote
Service A
live

What does this all mean?

Follow us: https://github.com/bbc

Questions?

Moltes gràcies!